Program Analysis Tools

Steven J Zeil

April 18, 2013
Outline
Analysis Tools

- **Static Analysis**
  - style checkers
  - data flow analysis
- **Dynamic Analysis**
  - Memory use monitors
  - Profilers
Analysis Tools and Compilers

Analysis tools, particularly static, share a great deal with compilers

- Need to parse code & perform limited static analysis
  - Generally working from ASTs
  - Some exceptions (working from object code or byte code)
- Data flow techniques originated in compiler optimization
Outline I
Abstract Syntax Trees

- Output of a language parser
  - Simpler than parse trees
- Generally viewed as a generalization of operator-applied-to-operands
Abstract Syntax Trees (cont.)

- ASTs can be applied to larger constructions than just expressions.
- In fact, generally reduce entire program or compilation unit to one AST.

```
a b
≥
a b
0
```

```
if
├──>
│   ├──:=
│   │   ├──a
│   │   └──a
│   └──:=
│       ├──a
│       └──b
```

```
Abstract Syntax Trees (cont.)

```plaintext
function
  paramList
    param
      a int
    param
      b int
  body
    if
      > := :=
        a b a a - a 0
        a b
```
Abstract Syntax Graphs

- Semantic analysis pairs uses of variables with declarations
- Transforming the AST into an ASG
Outline I
All data-flow information is obtained by propagating data flow markers through the program.

The usual markers are

- $d(x)$: a definition of variable $x$ (any location where $x$ is assigned a value)
- $r(x)$: a reference to $x$ (any location where the value of $x$ is used)
- $u(x)$: an undefinition of $x$ (any location where $x$ becomes undefined/illegal)
Propagation of Markers

For each node (basic block) in the control flow graph, we define

- $\text{gen}(n) =$ set of data-flow markers generated within node $n$.
- $\text{kill}(n) =$ set of data-flow markers killed within node $n$.
- $\text{in}(n) =$ set of data-flow markers entering node $n$ from elsewhere.
- $\text{out}(n) =$ set of data-flow markers leaving node $n$ to go elsewhere.

The basic data flow problem is to find $\text{in}()$ and $\text{out}()$ for each node given the control flow graph and the $\text{gen}()$ and $\text{kill}()$ sets for each node.
procedure SQRT (Q, A, B: in float; n_0
    X: out float);
    // Compute X = square root of Q,
    // given that A <= X <= B
    X1, F1, F2, H: float;
begin
    X1 := A;
    X2 := B;
    F1 := Q - X1**2
    H := X2 - X1;
    while (ABS(H) >= 0.001) loop
        F2 := Q - X2**2;
        H := -F2 * ((X2-X1)/(F2-F1));
        X1 := X2;
        X2 := X2 + H;
        F1 := F2
    end loop;
    X := (X1 + X2) / 2.;
end SQRT;
Reaching Definitions

A definition $d_i(x)$ reaches a node $n_j$ iff there exists a path from $n_i$ to $n_j$ on which $x$ is neither defined nor undefined.
The Reaching DF Problem

\[ \text{gen}(n) = \text{set of definitions occurring in } n \text{ and reaching the end of } n. \]

\[ \text{kill}(n) = \text{set of all definitions } d_i(x) \text{ in the CFG such that } x \text{ is defined or undefined within } n. \]

\[ \text{in}(n) = \bigcup_{m \in \text{pred}(n)} \text{out}(m) \]

\[ \text{out}(n) = (\text{in}(n) - \text{kill}(n)) \cup \text{gen}(n) \]
Program Analysis Tools
Data Flow Analysis

Sample Nodes

\[\begin{align*}
\text{gen}(n_0) &= \{d_0(Q), d_0(A), d_0(B)\} \\
\text{gen}(n_1) &= \{d_1(X1), d_1(X2), d_1(F1), d_1(H)\} \\
\text{gen}(n_2) &= \{\} \\
\text{gen}(n_3) &= \{d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1)\} \\
\text{gen}(n_4) &= \{d_4(X)\} \\
\text{gen}(n_5) &= \{\} 
\end{align*}\]
Sample Nodes (kill)

\[
\text{kill}(n_0) = \{ d_0(Q), d_0(A), d_0(B), d_1(X1), d_1(X2), \\
               d_1(F1), d_1(H), d_3(F2), d_3(H), d_3(X1), \\
               d_3(X2), d_3(F1), d_4(X) \}
\]

\[
\text{kill}(n_1) = \{ d_1(X1), d_1(X2), d_1(F1), d_1(H), d_3(H), \\
               d_3(X1), \}
\]

\[
\text{kill}(n_2) = \{\}
\]

\[
\text{kill}(n_3) = \{ d_1(X1), d_1(X2), d_1(F1), d_1(H), d_3(F2), \\
               d_3(H), d_3(X1), d_3(X2), d_3(F1) \}
\]

\[
\text{kill}(n_4) = \{ d_4(X) \}
\]

\[
\text{kill}(n_5) = \{ d_0(Q), d_0(A), d_0(B), d_1(X1), d_1(X2), \\
               d_1(F1), d_1(H), d_3(F2), d_3(H), d_3(X1), \\
               d_3(X2), d_3(F1) \}
\]
Solving for ReachingDefs

Solving iteratively, we start with $in(n) = out(n) = \{\}$, and propagate definitions.

First Iteration:

$in(0) = \{\}$

$out(0) = gen(0)$

$in(1) = gen(0)$

$out(1) = gen(0) \cup gen(1)$
Iteration 1 (cont.)

\[
\begin{align*}
in(2) & = \text{gen}(0) \cup \text{gen}(1) \\
\text{out}(2) & = \text{gen}(0) \cup \text{gen}(1) \\
in(3) & = \text{gen}(0) \cup \text{gen}(1) \\
\text{out}(3) & = \{d_0(Q), d_0(A), d_0(B), d_3(F2), d_3(H), \}
\{d_3(X1), d_3(X2), d_3(F1)\} \\
in(4) & = \text{gen}(0) \cup \text{gen}(1) \\
\text{out}(4) & = \text{gen}(0) \cup \text{gen}(1) \cup \{d_4(X)\} \\
in(5) & = \text{gen}(0) \cup \text{gen}(1) \cup \{d_4(X)\} \\
\text{out}(5) & = \{d_4(X)\}
\end{align*}
\]
Iteration 2

\[\begin{align*}
\text{in}(0) &= \text{unchanged} \\
\text{out}(0) &= \text{unchanged}
\end{align*}\]

\[\begin{align*}
\text{in}(1) &= \text{unchanged} \\
\text{out}(1) &= \text{unchanged}
\end{align*}\]

\[\begin{align*}
\text{in}(2) &= \text{gen}(0) \cup \text{gen}(1) \cup \{d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1)\} \\
\text{out}(2) &= \text{gen}(0) \cup \text{gen}(1) \cup \{d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1)\}
\end{align*}\]
Iteration 2 (cont.)

\[
in(3) = \text{gen}(0) \cup \text{gen}(1) \cup \{ d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1), \}
\]
\[
out(3) = \text{unchanged}
\]

\[
in(4) = \text{gen}(0) \cup \text{gen1} \cup \{ d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1), \}
\]
\[
out(4) = \text{gen}(0) \cup \text{gen1} \cup \{ d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1), d_4(X) \}
\]

\[
in(5) = \text{gen}(0) \cup \text{gen1} \cup \{ d_3(F2), d_3(H), d_3(X1), d_3(X2), d_3(F1), d_4(X) \}
\]
\[
out(5) = \text{unchanged}
\]
Data Flow Anomalies

The reaching definitions problem can be used to detect anomalous patterns that *may* reflect errors.

- *ur anomalies*: if an undefined definition of a variable *reaches* a reference of the same variable
- *dd anomalies*: if a definition of a variable *reaches* a definition of the same variable
- *du anomalies*: if a definition of a variable *reaches* an undefined definition of the same variable
Available Expressions

An expression $e$ is *available* at a node $n$ iff every path from the start of the program to $n$ evaluates $e$, and iff, after the last evaluation of $e$ on each such path, there are no subsequent definitions or undefinitions to the variables in $e$. 
The Available DF Problem

\( \text{gen}(n) = \) set of expressions evaluated in \( n \) containing no variables subsequently defined or undefined within \( n \).

\( \text{kill}(n) = \) set of all expressions in the program containing variables that are defined or undefined within \( n \).

\[
\text{in}(n) = \bigcap_{m \in \text{pred}(n)} \text{out}(m)
\]

\[
\text{out}(n) = (\text{in}(n) - \text{kill}(n)) \cup \text{gen}(n)
\]
Live Variables

A variable $x$ is *live* at node $n$ iff there exists a path starting at $n$ along which $x$ is used without prior redefinition.
The Live Variable DF Problem

\[ \text{gen}(n) = \text{set of variables used in } n \text{ without prior definition.} \]

\[ \text{kill}(n) = \text{set of variables defined within } n. \]

\[ \text{in}(n) = \text{gen}(n) \cup (\text{out}(n) - \text{kill}(n)) \]

\[ \text{out}(n) = \bigcup_{m \in \text{succ}(n)} \text{in}(m) \]
## Data Flow and Optimization

<table>
<thead>
<tr>
<th>Optimization Technique</th>
<th>Data-Flow Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Propagation</td>
<td>reach</td>
</tr>
<tr>
<td>Copy Propagation</td>
<td>reach</td>
</tr>
<tr>
<td>Elimination of Common Subexpressions</td>
<td>available</td>
</tr>
<tr>
<td>Dead Code Elimination</td>
<td>live, reach</td>
</tr>
<tr>
<td>Register Allocation</td>
<td>live</td>
</tr>
<tr>
<td>Anomaly Detection</td>
<td>reach</td>
</tr>
<tr>
<td>Code Motion</td>
<td>reach</td>
</tr>
</tbody>
</table>
Outline I
Lint

Perhaps the first such tool to be widely used, lint (1979) became a staple tool for C programmers. Combines static analysis with style recommendations, e.g.,

- data flow anomalies
- potential arithmetic overflow
  - e.g., storing an int calculation in a char
- conditional statements with constant values
- potential = versus == confusion
Is there room for lint-like tools?

- **lint** was a response, in part, to the weak capabilities of early C compilers.
- Much of what **lint** does is now handled by optimizing compilers.
  - However compilers seldom do cross-module or even cross-function analysis.
FindBugs

- Open source project from U.Md.
- Works on compiled Java bytecode
- Sample report
- Can be run via
  - GUI
  - ant
  - Eclipse
  - maven
What Bugs does FindBugs Find?

- “Bugs” categorized as
  - Correctness bug: an apparent coding mistake
  - Bad Practice: violations of recommended coding practices.
  - Dodgy: code that is “confusing, anomalous, or written in a way that leads itself to errors”

- Bugs are also given “priorities” (p1, p2, p3 from high to low)

- Bug list
PMD

- PMD, source analysis for Java, JavaScript, XSL
  - CPD, “copy-paste-detector” for many programming languages
- Works on source code
- Sample reports (PMD & CPD)
- Can be run via bii ant
  - maven
  - eclipse
PMD Reports

- Configured by selecting “rule set” modules
  - Otherwise, appears to lack categories & priorities
- Cross reference to source location
Reverse Compilers

a.k.a. “uncompilers”

- Generate source code from object code
- Originally clunky & more of a curiosity than usable tools
  - Improvements based on
    - “deep” knowledge of compilers (aided by increasingly limited field of available compilers)
    - Information-rich object codes (e.g., Java bytecode formats)
- Legitimate uses include
  - reverse-engineering
  - generating input for source-based analysis tools
- But also great tools for plagiarism
Java and Decompilation

- Java is a particularly friendly field for decompilers
  - Rich object code format
  - Nearly monopolistic compiler suite
- Options for “protecting” programs compiled in Java:
  - **gjc**: compile into native code with a far less popular compiler
  - obfuscators
Java Obfuscators

Work by a combination of

- Renaming variables, functions, and classes to meaningless, innocuous, and very similar name sets
  - Challenge is to preserve those names of entry points needed to execute a program or applet or make calls upon a library’s public API
  - Stripping away debugging information (e.g., source code file names and line numbers associated with blocks of code)
  - Applying optimization techniques to reduce code size while also confusing the object-to-source mapping

Example, yguard
Outline 1
Dynamic Analysis Tools

Not all useful analysis can be done statically

- Profiling
- Memory leaks, corruption, etc.
- Data structure abuse
Abusing Data Structures

- Traditionally, the C++ standard library does not check for common abuses such as over-filling and array or accessing non-existent elements
  - Various authors have filled in with “checking” implementations of the library for use during testing and debugging
- In a sense, the `assert` command of C++ and Java is the language’s own extension mechanism for such checks.
Memory Abuse

- Pointer errors in C++ are both common and frustrating
  - Traditionally unchecked by standard run-time systems
- Monitors can be added to help catch these
  - In C++, link in a replacement for malloc & free
How to Catch Pointer Errors

- Use *fenceposts* around allocated blocks of memory
  - check for unchanged fenceposts to detect over-writes
  - Check for fenceposts before a delete to detect attempts to delete addresses other than the start of an allocated block
- Add tracking info to allocated blocks indicating location of the allocation call
  - Scan heap at end of program for unrecovered blocks of memory
  - Report on locations from which those were allocated
- Add a “freed” bit to allocated blocks that is cleared when first allocated and set when the block is freed
  - Detect when a block is freed twice
Memory Analysis Tools

- Purify is a well-known commercial (pricey) tool
- At the other end of the spectrum, LeakTracer is a small, simple, but capable open source package that I’ve used for many years
  - Works with gcc/g++/gdb compiler suite
  
  leaktracer.listing
Profilers

Profilers provide info on where a program is spending most of its execution time

- May express measurements in
  - Elapsed time
  - Number of executions

- Granularity may be at level of
  - functions
  - individual lines of code

- Measurement may be via
  - Probes inserted into code
  - Statistical sampling of CPU program counter register
Profiling Tools

- **gprof** for C/C++, part of the GNU compiler suite
  - Refer back to earlier lesson on statement and branch coverage
  - **gprof** is, essentially, the generalization of **gcov**

- **jvisualm** for Java, part of the Java SDK

- Provides multiple monitoring tools, including both CPU and memory profiling